# Causality is simple

Carlos Baquero

Univ. Minho & INESC TEC
Portugal

Berlin Buzzwords 2016

HASLab
HIGH-ASSURANCE SOFTWARE LABORATORY

Universidade do Minho

INESCTEC
TECHNOLOGY & SCIENCE
ASSOCIATE LABORATORY
COORDINATED BY
INESCPORTO
PORTUGAL

# Causality is (moderatly) simple

Carlos Baquero

Univ. Minho & INESC TEC
Portugal

Berlin Buzzwords 2016

# Why should I care?

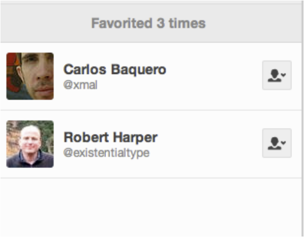
# Everything is distributed

*"Distributed systems once were the territory of computer science Ph.D.s and software architects tucked off in a corner somewhere. Thats no longer the case."*

(2014 http://radar.oreilly.com/2014/05/everything-is-distributed)

# Anomalies in Distributed Systems

- A buzz, you have a new message, but message is not there yet
- Remove your boss from a group, post to it and he sees posting
- In LWW + bad clocks, read a value and cannot overwrite it
- Assorted inconsistencies



# Time

Can't we use time(stamps) to fix it?

# Time

The problem with time is that

**Distributed Computing**

March 10, 2015
**Volume 13, issue 3**

PDF

## There is No Now

### Problems with simultaneity in distributed systems

**Justin Sheehy**

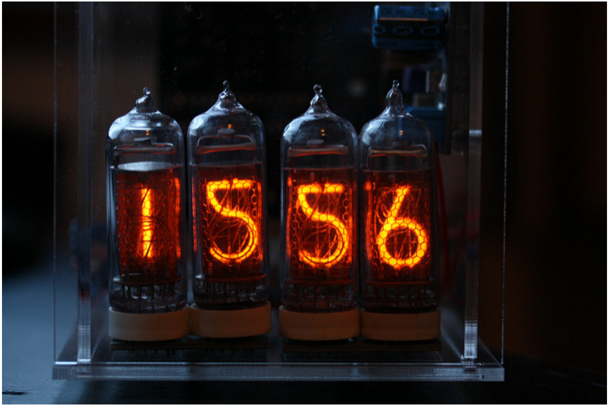
"Now."

The time elapsed between when I wrote that word and when you read it was at least a couple of weeks. That kind of delay is one that we take for granted and don't even think about in written media.

(2015 http://queue.acm.org/detail.cfm?id=2745385)

# Light speed is causality speed

# Time is local



# Time needs memory

(Before)

# Time needs memory

(After)

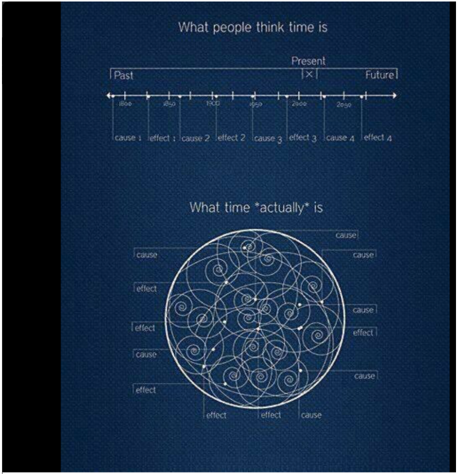
# Time needs memory

(Before)

# Time needs memory

(After)

# Its complicated . . .




**VIP Coaching** @coaching_vip · Apr 29

A different perception of time ❤️ #Causality is #Reality

2

# Causality

Operating Systems

R. Stockton Gaines Editor

## Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport
Massachusetts Computer Associates, Inc.

**The concept of one event happening before another in a distributed system is examined, and is shown to define a partial ordering of the events. A distributed algorithm is given for synchronizing a system of logical clocks which can be used to totally order the events. The use of the total ordering is illustrated with a method for solving synchronization problems. The algorithm is then specialized for synchronizing physical clocks, and a bound is derived on how far out of synchrony the clocks can become.**

**Key Words and Phrases: distributed systems, computer networks, clock synchronization, multiprocess systems**

**CR Categories: 4.32, 5.29**

(1978 http://amturing.acm.org/p558-lamport.pdf)

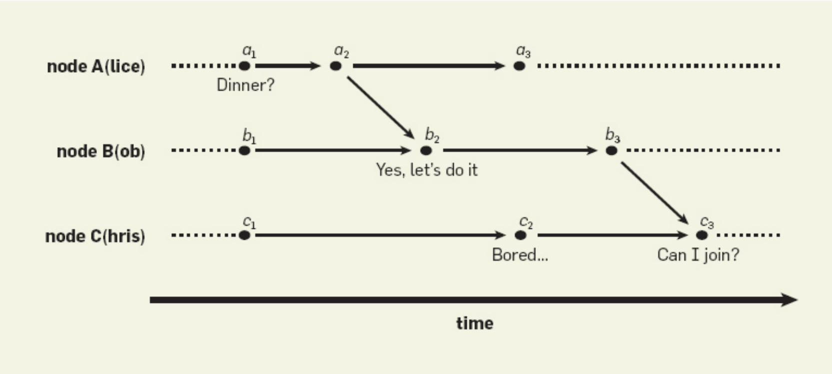
# Causality

## A social interaction

- Alice decides to have dinner.
- She tells that to Bob and he agrees.
- Meanwhile Chris was bored.
- Bob tells Chris and he asks to join for dinner.

# Causality relation

Events get unique tags (dots), by place name and growing counter




Causally: "Alice wants dinner" $\parallel$ "Chris is bored"
Timeline: "Alice wants dinner" $\rightarrow$ "Chris is bored"

# Causality relation

How to track it?

# Causality relation

How to track it? Maybe read Vector Clock entry in Wikipedia?

## Partial ordering property [ edit ]

Vector clocks allow for the partial causal ordering of events. Defining the following:

- $VC(x)$ denotes the vector clock of event $x$, and $VC(x)_z$ denotes the component of that clock for process $z$.
- $VC(x) < VC(y) \iff \forall z[VC(x)_z \leq VC(y)_z] \wedge \exists z'[VC(x)_{z'} < VC(y)_{z'}]$
  - In English: $VC(x)$ is less than $VC(y)$, if and only if $VC(x)_z$ is less than or equal to $VC(y)_z$ for all process indices $z$, and at least one of those relationships is strictly smaller (that is, $VC(x)_{z'} < VC(y)_{z'}$).
- $x \to y$ denotes that event $x$ happened before event $y$. It is defined as: if $x \to y$, then $VC(x) < VC(y)$

Properties:

- If $VC(a) < VC(b)$, then $a \to b$
- Antisymmetry: if $VC(a) < VC(b)$, then $\neg VC(b) < VC(a)$
- Transitivity: if $VC(a) < VC(b)$ and $VC(b) < VC(c)$, then $VC(a) < VC(c)$ or if $a \to b$ and $b \to c$, then $a \to c$

Relation with other orders:

- Let $RT(x)$ be the real time when event $x$ occurs. If $VC(a) < VC(b)$, then $RT(a) < RT(b)$
- Let $C(x)$ be the Lamport timestamp of event $x$. If $VC(a) < VC(b)$, then $C(a) < C(b)$

(2015 https://en.wikipedia.org/wiki/Vector_clock)

Maybe start with something simpler: **Causal histories**

# Causal histories

**Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail**

Reinhard Schwarz
Department of Computer Science, University of Kaiserslautern,
P.O. Box 3049, D - 67653 Kaiserslautern, Germany
schwarz@informatik.uni-kl.de

Friedemann Mattern
Department of Computer Science, University of Saarland
Im Stadtwald 36, D - 66041 Saarbrücken, Germany
mattern@cs.uni-sb.de

***Abstract***: *The paper shows that characterizing the causal relationship between significant events is an important but non-trivial aspect for understanding the behavior of distributed programs. An introduction to the notion of causality and its relation to logical time is given; some fundamental results concerning the characterization of causality are presented. Recent work on the detection of causal relationships in distributed computations is surveyed. The issue of observing distributed computations in a causally consistent way and the basic problems of detecting global predicates are discussed. To illustrate the major difficulties, some typical monitoring and debugging approaches are assessed, and it is demonstrated how their feasibility is severely limited by the fundamental problem to master the complexity of causal relationships.*

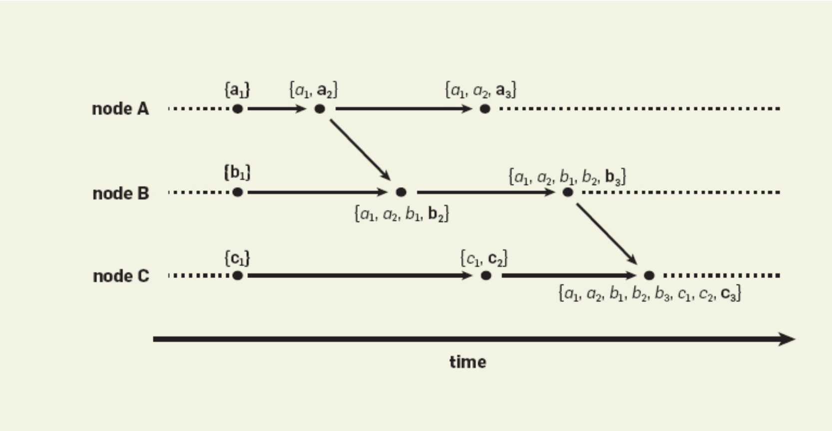
***Keywords:*** *Distributed Computation, Causality, Distributed System, Causal Ordering, Logical Time, Vector Time, Global Predicate Detection, Distributed Debugging, Timestamps*

(1994 https://www.vs.inf.ethz.ch/publ/papers/holygrail.pdf)

# Causal histories

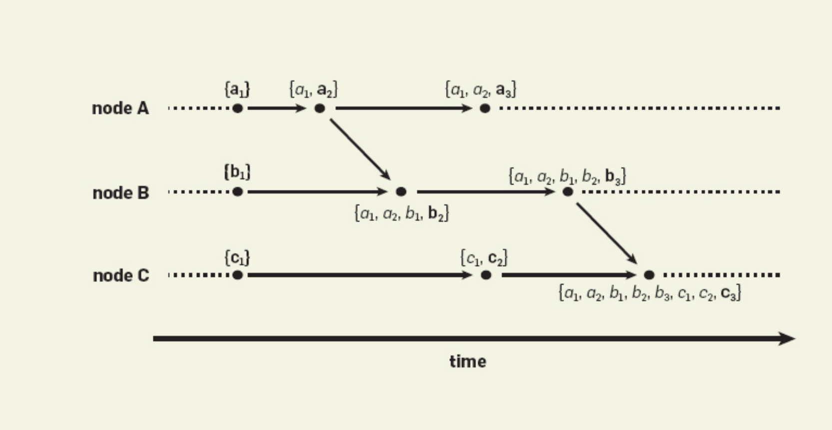
- Collect memories as sets of unique events (dots)
- Set inclusion explains causality
  - $\{a_1, b_1\} \subset \{a_1, a_2, b_1\}$
- You are in my past if I know your history
- If we don't know each other's history, we are concurrent
- If our histories are the same, we are the same

# Causal histories



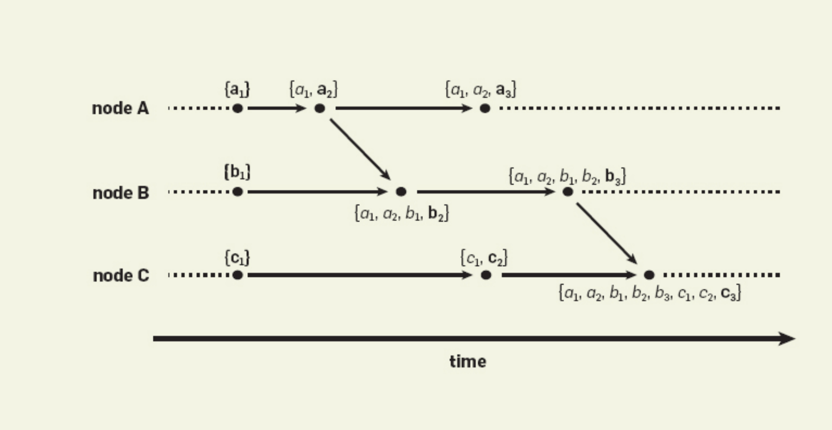
# Causal histories

## Message reception




Receive $\{a_1, a_2\}$ at node $b$ with $\{b_1\}$ yields $\{b_1\} \cup \{a_1, a_2\} \cup \{b_2\}$

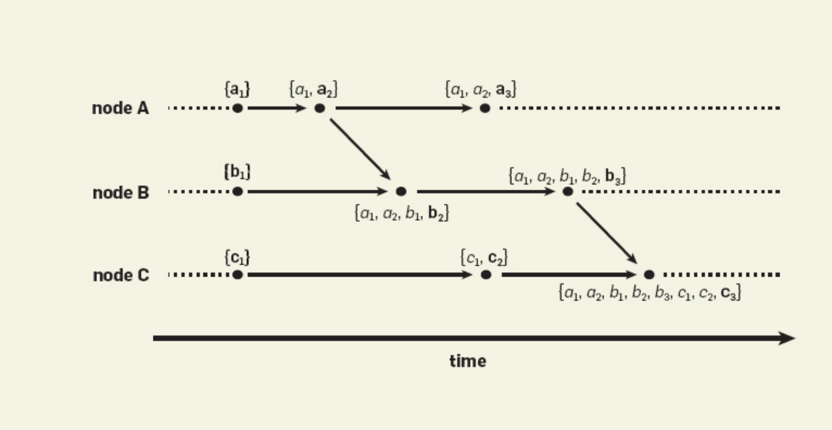
# Causal histories
## Causality check




Check $\{a_1, a_2\} \to \{a_1, a_2, b_1, b_2\}$ iff $\{a_1, a_2\} \subset \{a_1, a_2, b_1, b_2\}$

# Causal histories

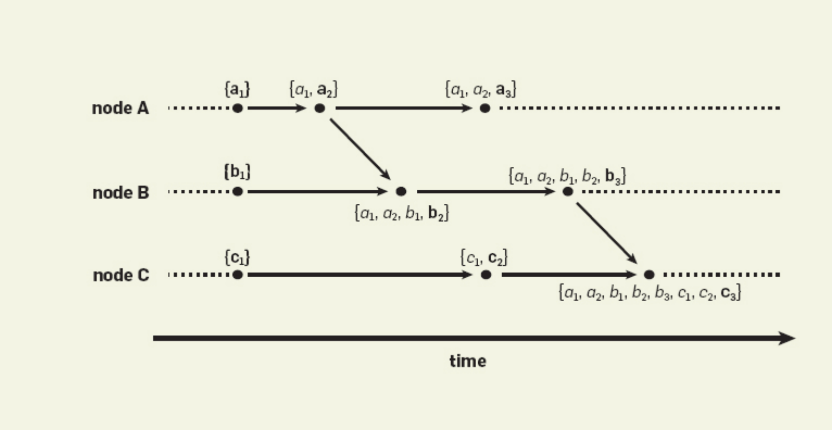
## Causality check




Check $\{a_1, \mathbf{a_2}\} \rightarrow \{a_1, a_2, b_1, \mathbf{b_2}\}$ iff $\{a_1, \mathbf{a_2}\} \subset \{a_1, a_2, b_1, \mathbf{b_2}\}$

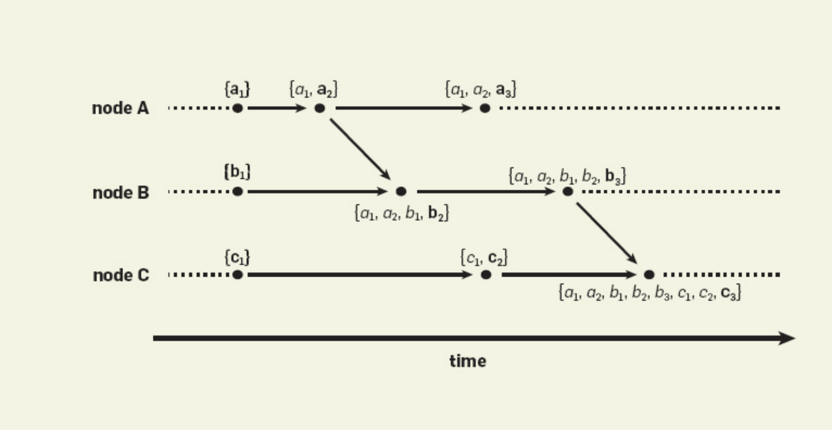
# Causal histories

## Faster causality check



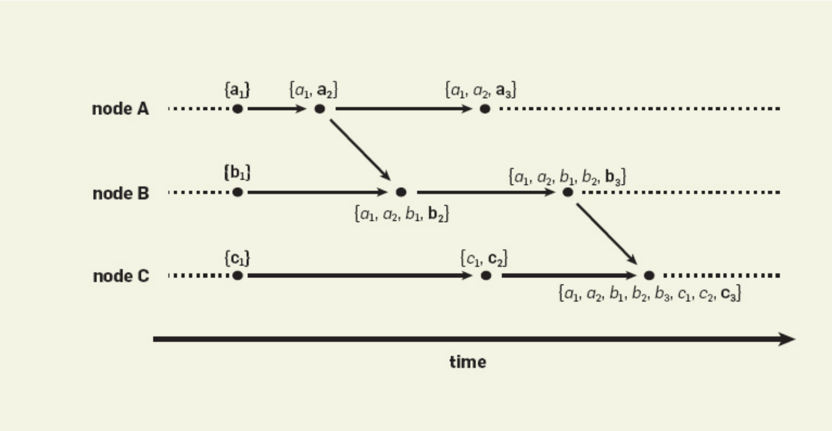

Check $\{a_1, \mathbf{a_2}\} \rightarrow \{a_1, a_2, b_1, \mathbf{b_2}\}$ iff $\mathbf{a_2} \in \{a_1, a_2, b_1, \mathbf{b_2}\}$

# Causal histories



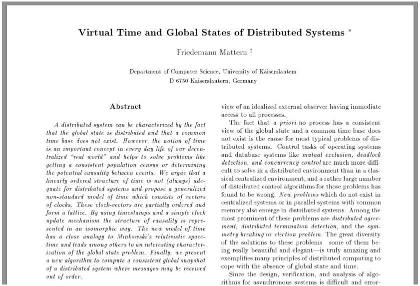

Note $e_n \in \{e_1 \ldots e_n, f_1 \ldots\}$ implies $\{e_1 \ldots e_n\} \subset \{e_1 \ldots e_n, f_1 \ldots\}$

# Causal histories




Lots of redundancy than can be compressed

# Vector clocks



## Virtual Time and Global States of Distributed Systems *

Friedemann Mattern †

Department of Computer Science, University of Kaiserslautem
D 6750 Kaiserslautern, Germany

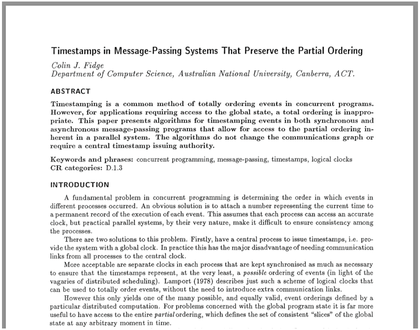
### Abstract

*A distributed system can be characterized by the fact that the global state is distributed and that a common time base does not exist. However, the notion of time is an important concept in every day life of our decentralized "real world" and helps to solve problems like getting a consistent population census or determining the potential causality between events. We argue that a linearly ordered structure of time is not (always) adequate for distributed systems and propose a generalized non-standard model of time which consists of vectors of clocks. These clock-vectors are partially ordered and form a lattice. By using timestamps and a simple clock update mechanism the structure of causality is represented in an isomorphic way. The new model of time has a close analogy to Minkowski's relativistic space-time and leads among others to an interesting characterization of the global state problem. Finally, we present a new algorithm to compute a consistent global snapshot of a distributed system where messages may be received out of order.*

view of an idealized external observer having immediate access to all processes.

The fact that *a priori* no process has a consistent view of the global state and a common time base does not exist is the cause for most typical problems of distributed systems. Control tasks of operating systems and database systems like *mutual exclusion, deadlock detection, and concurrency control* are much more difficult to solve in a distributed environment than in a classical centralized environment, and a rather large number of distributed control algorithms for those problems has found to be wrong. *New problems* which do not exist in centralized systems or in parallel systems with common memory also emerge in distributed systems. Among the most prominent of these problems are *distributed agreement, distributed termination detection*, and the *symmetry breaking* or *election problem*. The great diversity of the solutions to these problems some of them being really beautiful and elegant—is truly amazing and exemplifies many principles of distributed computing to cope with the absence of global state and time.

Since the design, verification, and analysis of algorithms for asvnchronous svstems is difficult and error-



## Timestamps in Message-Passing Systems That Preserve the Partial Ordering

*Colin J. Fidge*
*Department of Computer Science, Australian National University, Canberra, ACT.*

### ABSTRACT

**Timestamping is a common method of totally ordering events in concurrent programs. However, for applications requiring access to the global state, a total ordering is inappropriate. This paper presents algorithms for timestamping events in both synchronous and asynchronous message-passing programs that allow for access to the partial ordering inherent in a parallel system. The algorithms do not change the communications graph or require a central timestamp issuing authority.**

**Keywords and phrases:** concurrent programming, message-passing, timestamps, logical clocks
**CR categories:** D.1.3

### INTRODUCTION

A fundamental problem in concurrent programming is determining the order in which events in different processes occurred. An obvious solution is to attach a number representing the current time to a permanent record of the execution of each event. This assumes that each process can access an accurate clock, but practical parallel systems, by their very nature, make it difficult to ensure consistency among the processes.

There are two solutions to this problem. Firstly, have a central process to issue timestamps, i.e. provide the system with a global clock. In practice this has the major disadvantage of needing communication links from all processes to the central clock.

More acceptable are separate clocks in each process that are kept synchronised as much as necessary to ensure that the timestamps represent, at the very least, a *possible* ordering of events (in light of the vagaries of distributed scheduling). Lamport (1978) describes just such a scheme of logical clocks that can be used to totally order events, without the need to introduce extra communication links.

However this only yields one of the many possible, and equally valid, event orderings defined by a particular distributed computation. For problems concerned with the global program state it is far more useful to have access to the entire *partial* ordering, which defines the set of consistent "slices" of the global state at any arbitrary moment in time.

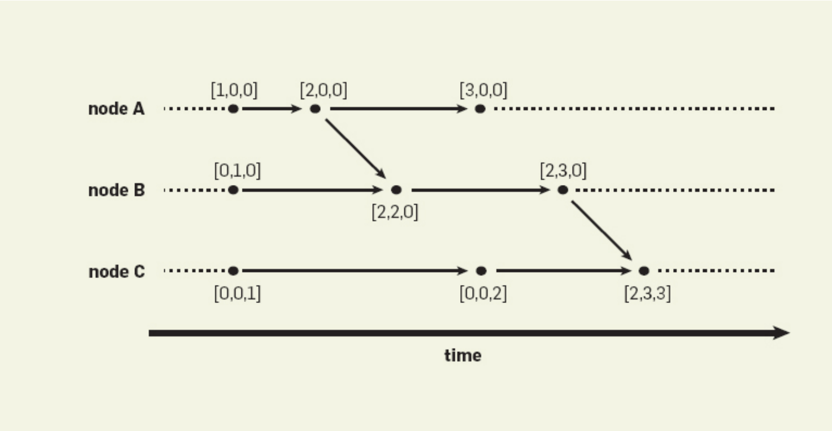
1988 (https://www.vs.inf.ethz.ch/publ/papers/VirtTimeGlobStates.pdf)
(http://zoo.cs.yale.edu/classes/cs426/2012/lab/bib/fidge88timestamps.pdf)

- $\{a_1, a_2, b_1, b_2, b_3, c_1, c_2, c_3\}$
- $\{a \mapsto 2, b \mapsto 3, c \mapsto 3\}$

Finally a vector, since $\langle a, b, c \rangle$ is a continuous sequence

- $[2, 3, 3]$

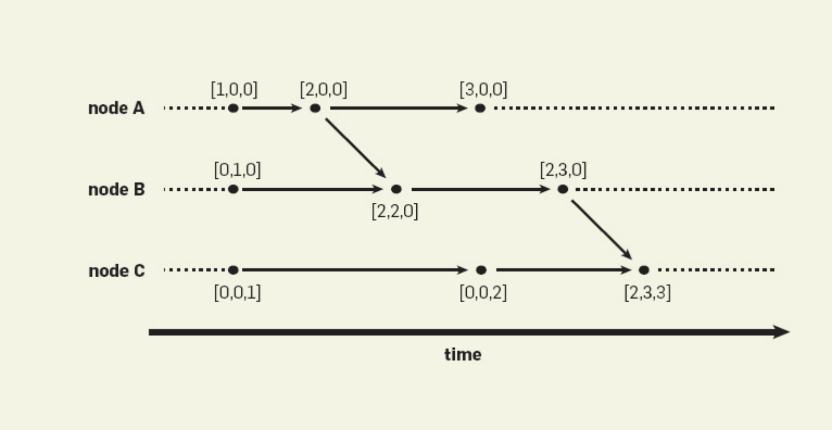
# Vector clocks



# Vector clocks

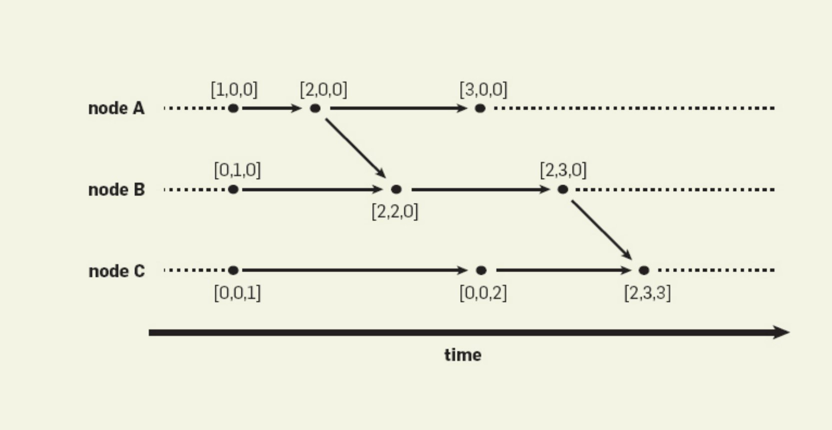
## Message reception

Set union becomes **join** $\sqcup$ by point-wise maximum in vectors



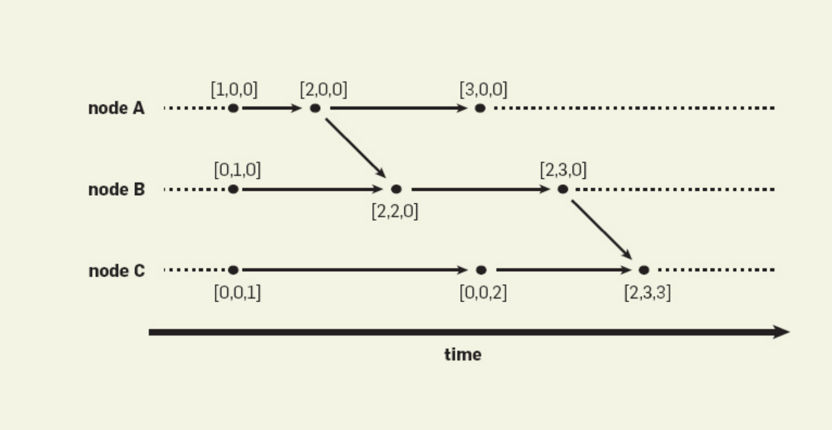

Receive $[2,0,0]$ at $b$ with $[0,1,0]$ yields $\text{inc}_b(\sqcup([2,0,0],[0,1,0]))$

# Vector clocks
## Causality check




Check $[2, 0, 0] \rightarrow [2, 2, 0]$ iff point-wise check $2 \leq 2, 0 \leq 2, 0 \leq 0$

# Vector clocks
## Faster causality check




Check $[\mathbf{2}, 0, 0] \rightarrow [2, \mathbf{2}, 0]$ iff $\mathbf{2} \leq 2$

# Vector clocks with dots

## Decouple dot of last event

- Not that easy to code **bold** in a PL
- $[\mathbf{2}, 0, 0]$ becomes $[1, 0, 0]a_2$
- $[2, \mathbf{2}, 0]$ becomes $[2, 1, 0]b_2$
- Now the causal past excludes the event itself
- Check $[\mathbf{2}, 0, 0] \rightarrow [2, \mathbf{2}, 0]$?
- Check $[1, 0, 0]a_2 \rightarrow [2, 1, 0]b_2$ iff dot $a_2$ index $_2 \leq 2$

# Registering relevant events

- Not always important to track all events
- Track only change events in data replicas
- Applications in:
  - File-Systems
  - Databases
  - Version Control

# Dynamo

Causally tracking of write/put operations

**Dynamo: Amazon's Highly Available Key-value Store**

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

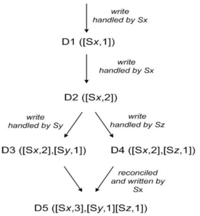



**Figure 3: Version evolution of an object over time.**

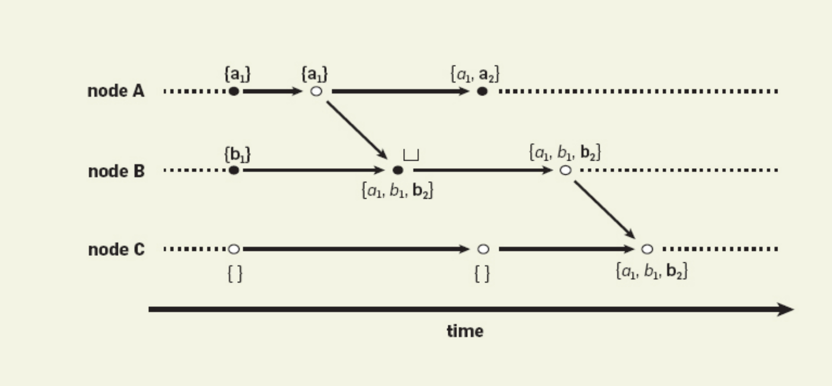
object. In practice, this is not likely because the writes are usually handled by one of the top N nodes in the preference list. In case of network partitions or multiple server failures, write requests may be handled by nodes that are not in the top N nodes in the preference list causing the size of vector clock to grow. In these scenarios, it is desirable to limit the size of vector clock. To this end, Dynamo employs the following clock truncation scheme: Along with each (node, counter) pair, Dynamo stores a timestamp that indicates the last time the node updated the data item. When the number of (node, counter) pairs in the vector clock reaches a threshold (say 10), the oldest pair is removed from the clock. Clearly, this truncation scheme can lead to inefficiencies in reconciliation as the descendant relationships cannot be derived accurately. However, this problem has not surfaced in production and therefore this issue has not been thoroughly investigated.

**4.5 Execution of get () and put () operations**

Any storage node in Dynamo is eligible to receive client get and put operations for any key. In this section, for sake of simplicity, we describe how these operations are performed in a failure-free environment and in the subsequent section we describe how read

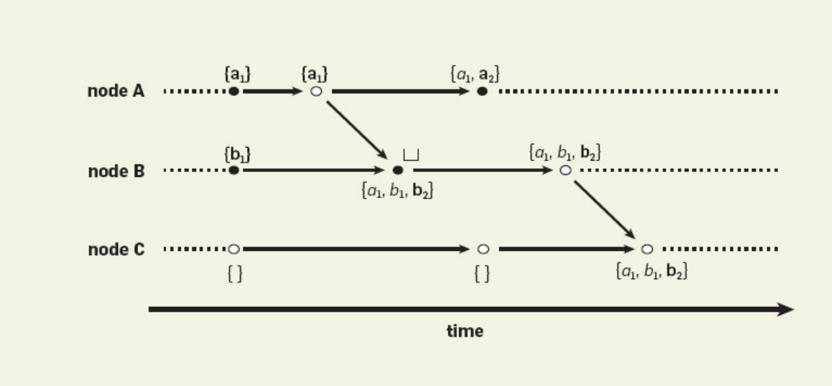
(http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf)

# Causal histories with only some relevant events



# Causal histories with only some relevant events

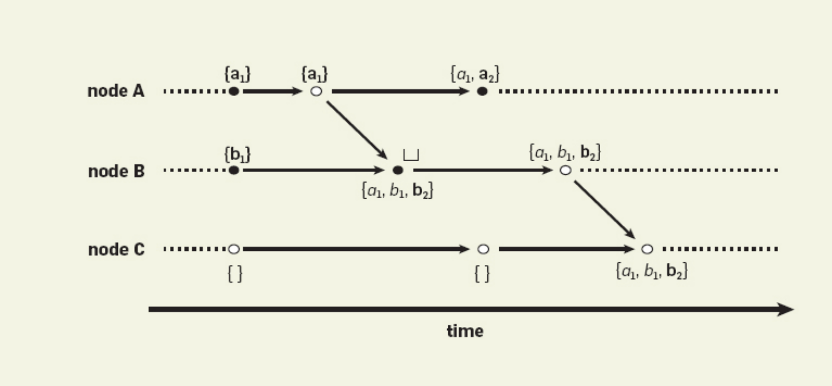
Relevant events are marked with a • and get an unique tag/dot




Other events get a ∘ and don't add to history

# Causal histories with only some relevant events

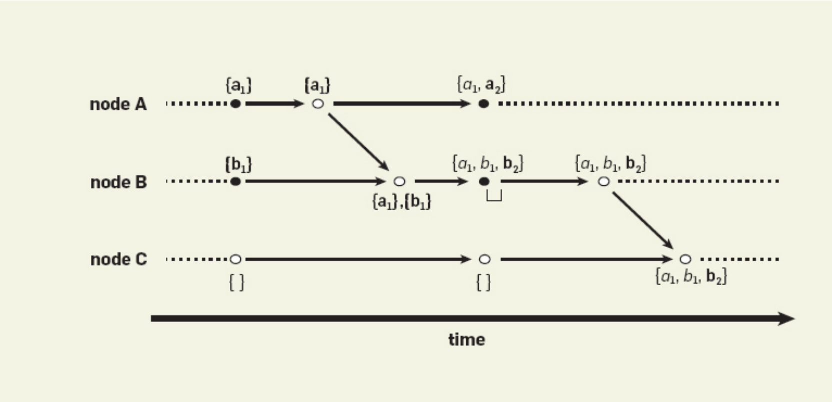
Concurrent states $\{a_1\} \parallel \{b_1\}$ lead to a • marked merge on join




Causally dominated state $\{\} \rightarrow \{a_1, b_1, b_2\}$ is simply replaced

# Causal histories with versions not immediately merged

Versions can be collected and merge deferred




Only when merging a new • is needed to reflect the state change

# Version vectors

240 IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. SE-9, NO. 3, MAY 1983

## Detection of Mutual Inconsistency in Distributed Systems

D. STOTT PARKER, JR., GERALD J. POPEK, GERARD RUDISIN, ALLEN STOUGHTON,
BRUCE J. WALKER, EVELYN WALTON, JOHANNA M. CHOW,
DAVID EDWARDS, STEPHEN KISER, AND CHARLES KLINE

*Abstract*—Many distributed systems are now being developed to provide users with convenient access to data via some kind of communications network. In many cases it is desirable to keep the system functioning even when it is partitioned by network failures. A serious problem in this context is how one can support redundant copies of resources such as files (for the sake of reliability) while simultaneously monitoring their mutual consistency (the equality of multiple copies). This is difficult since network failures can lead to inconsistency, and disrupt attempts at maintaining consistency. In fact, even the *detection* of inconsistent copies is a nontrivial problem. Naive methods either 1) compare the multiple copies entirely or 2) perform simple tests which will diagnose some consistent copies as inconsistent. Here a new approach, involving *version vectors* and *origin points*, is presented and shown to detect single file, multiple copy mutual inconsistency effectively. The approach has been used in the design of *LOCUS*, a local network operating system at UCLA.

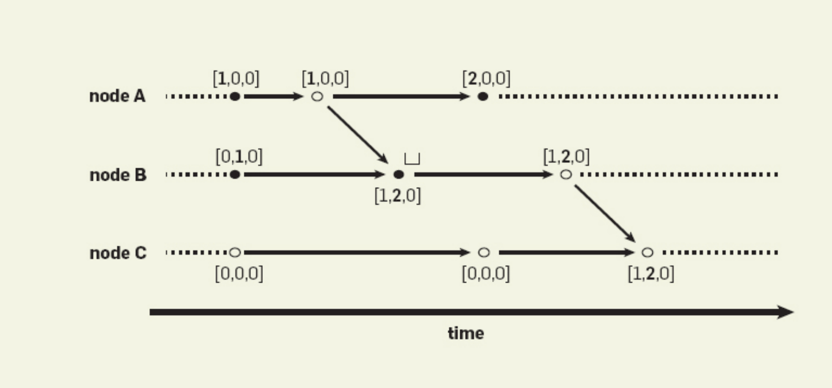
*Index Terms*—Availability, distributed systems, mutual consistency, network failures, network partitioning, replicated data.

multiple copies of a file exist, the system must ensure the *mutual consistency* of these copies: when one copy of the file is modified, all must be modified correspondingly before an independent access can take place.

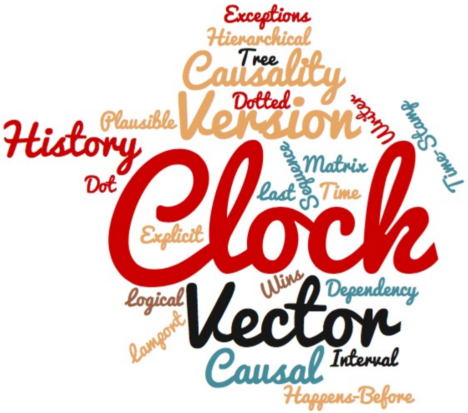
Much has been written about the problem of maintaining consistency in distributed systems, ranging from *internal* consistency methods (ways to keep a single copy of a resource looking consistent to multiple processes attempting to access it concurrently) to various ingenious updating algorithms which ensure mutual consistency [1], [2], [6], [8], [16], etc. We concern ourselves here with mutual consistency in the face of *network partitioning*, i.e., the situation where various sites in the network cannot communicate with each other for some length of time due to network failures or site crashes. This is a very real problem in most networks. For example, even in the Ethernet [10], gateways can be inoperative for significant lengths of time, while the Ether segments they normally connect operate correctly.

(1983 http://zoo.cs.yale.edu/classes/cs422/2013/bib/parker83detection.pdf)

# Version vectors



# Causality tracking mechanisms



# Closing notes

- Causality is important because time is limited
- Causality is about memory of relevant events
- Causal histories are very simple encodings of causality
- Practical mechanisms (VCs, VVs, DVVs) do efficient encoding

# Closing notes

When faced with a new design or mechanism: Try to think and translate back to a simple causal history.

**Programming Languages**

April 12, 2016
**Volume 14, issue 1**

PDF

## Why Logical Clocks are Easy

### Sometimes all you need is the right language.

**Carlos Baquero and Nuno Preguiça**

(2016 http://queue.acm.org/detail.cfm?id=2917756)

# Closing notes, questions?

When faced with a new design or mechanism: Try to think and translate back to a simple causal history.

**Programming Languages**

April 12, 2016
**Volume 14, issue 1** PDF

## Why Logical Clocks are Easy

**Sometimes all you need is the right language.**

**Carlos Baquero and Nuno Preguiça**

(2016 http://queue.acm.org/detail.cfm?id=2917756)

Email: cbm@di.uminho.pt, Twitter: @xmal, GitHub: CBaquero